Automatic Refactoring by Simulation of Multiple Inheritance in Java

Douglas Lyon, Ph.D.
Chair, Computer Engineering Dept.
Fairfield University, Fairfield CT 06430
lyon@docjava.com

The Perfect bureaucrat is the man
who manages to make no decisions
and escapes all responsibility.

– Justin Brooks Atkinson

Summary

This paper shows a technique that enables the generation of proxy classes in an automatic manner. The model and implementation extends production programming to support fast and automatic prototyping of proxies and interfaces able to simulate multiple inheritance and refactor legacy systems, even when faced with missing source code. Advantages of the automatic synthesis of a proxy class include: compile-time type checking, speed of execution, automatic disambiguation (name space collision resolution) and ease of maintenance.

The approach generates Java source that does method forwarding and creates *interfaces* as a means to achieve polymorphism. Disambiguation can be automatic, semi-automatic or manual. The forwarding code (i.e., proxy) evolves into an *adapter* as the delegates change their specification. This protects client classes from change. The interface and proxy code are generated automatically via *reflection*. This type of simulation of multiple inheritance was previously available only to Java programmers who performed manual delegation or who made use of dynamic proxies. The technique has been applied at a major aerospace corporation.

Key words: software reverse engineering; reverse engineering; reflection; delegation; Java; automatic code generation

1. Introduction

Refactoring is defined as “changing a system to improve its internal structure without altering its external behavior”. Often we are faced with legacy code that has to be refactored. Refactoring is typically done in order to improve some feature, such as design or readability. Refactoring is a key approach for improving object-oriented software systems [Tichelaar]. Sometimes the code has no source available and/or the design is poor. Sometimes a large number of dependencies between classes can complicate analysis [Korman]. While consulting for one aircraft manufacturer we were faced with Java code written by FORTRAN programmers (dubbed *JavaTran*). Over time, maintenance changed the original program structure and specifications. Additionally, the specifications had not been maintained. This is a common problem in industry [Postema].

According to one definition, delegation uses a receiving instance that forwards messages (or invocations) to its delegate(s). This is sometimes called a consultation [Kniesel]. Variations on this theme give rise to several of the so-called *design patterns*. For example, if methods are forwarded without change to the interface, then you have an example of the *proxy pattern*. If you simplify the interface with a subset of methods to a set of delegates, then you have a *facade pattern*. If you compensate for changes (i.e., deprecations) in the delegates, and keep the client classes seeing the same contract, then you have the *adapter pattern*. If you add responsibilities to the proxy class, then you have the *decorator pattern* [Gamma 1995]. Thus, we define *static delegation* as a compile-time type-safe message forwarding from a proxy class to some delegate(s).

Compare this to the definition given to use by Lieberman [Lie 1986]. With Lieberman-delegation the communications pattern is decided at run-time. This is

rather more flexible than the static delegation, so we define it as *dynamic delegation*. Thus, compile-time checks are not performed and the message forwarding is not type-safe. In JDK1.3 dynamic delegation is more automatic (i.e., it is Lieberman-style). Using the JDK 1.3 version of dynamic delegation you build a proxy object from the reflection API.

Our goal was to refactor the code in a *type-safe* way, without having to rewrite it. It is well known that improper refactoring can break subtle properties in a system. As a result, refactoring is generally followed by a testing phase [Katoaka]. To eliminate the testing phase after refactoring we have created an automatic means of generating proxy classes. These proxy classes are like *toolkits* that provide access to domain-specific frameworks [Gamma 1995]. The result is a stable interface to a large collection of methods in a single proxy class, rather than a weak coupling to many instances of several different classes. Since old code is unchanged (and even unneeded!) in our system, the new code can be treated as a facade for interfacing to the legacy code. Initially we wrote the proxies manually using a process we call *manual static delegation*. In manual static delegation, an instance is passed to a proxy class as a parameter. A programmer writes *wrapper* code that *delegates* to the contained instance. The code that contains the wrapper code is called the *proxy* class. The code that contains the implementation code is called the *delegate*. For example:

```
final class Movable {
      int x = 0;
      int y = 0;

      public void move(int _x, int _y) {
        x = _x;
        y = _y;
      }
}
```

If we want to add a feature to the *Movable* class we might subclass it. However, this is prevented because the class is *final*. We might be tempted to modify the

Movable class, however, source code might not be available. For example, suppose we want a *MovableMammal*:

```
class Mammal {
      public boolean isHairy() {
        return true;
      }
}
```

Our manually written delegation code follows:

```
public class MovableMammal {
      Mammal m;
      Movable mm;

      MovableMammal(Mammal _m, Movable _mm) {
        m = _m;
        mm = _mm;
      }

      public void move(int x, int y) {
        mm.move(x,y);
      }
      public void isHairy() {
        return m.isHairy();
      }
}
```

The automatic static proxy delegation generates code, like the *MovableMammal* class, *automatically*.

There are several alternatives to automatic proxy delegation for reusing implementations. For example, we can:

1. Deepen the subclass for processing data. This is the solution I took with *Image Processing in Java* [Lyon 1999]. Each chapter built another subclass until the classes were 9 levels deep. Subclassing is not always the best way to extend the functionality of a class. While subclassing is an object-oriented design technique, lack of multiple inheritance keeps Java from scaling this approach to large programs. Another approach is called *delegation*.

2. *Delegation*: Keep adding references to *helper* classes that can process the data, then delegate to the other classes for the implementation. Imagine if you needed to run a country. All requests from the citizens go to the president. Then the president makes a phone call, and delegates the request to the right person. The president acts in the role of the proxy class, dispatching to the correct implementor of a task. Delegation has long been thought of as a generalization of inheritance (a point of view with which there is disagreement) [Aksit 1991] [Bracha].
3. *Multiple Inheritance:* The exclusion of multiple inheritance from Java is a design decision that forces people into *doing what is good for them.* This is exactly what Stroustrup says that he tried to avoid in the design decisions that he made when creating C++ [Stro 1994]. Grady Booch has said that "Multiple inheritance is like a parachute; you don't need it very often, but when you do it is essential" [Booch 1991]. In contrast it has also been said that multiple inheritance is an inessential programming idiom [Compagnoni]. The multiple inheritance debates appear to be devoid of solid data. Programmers that use Java are innocent victims of the design decision to leave out multiple-inheritance [Stro 1987].

Generally, inheritance enables shared behavior. Some have argued that subtyping (i.e, the multiple-inheritance of interfaces in Java) is not inheritance. In fact, our approach divorced inheritance from subtyping creating new proxy classes with new interfaces. There is even disagreement on the appropriate semantics for multiple inheritance [Bracha] [Compagnoni]. Stroustrup say that multiple inheritance is the ability of a class to have more than one base class (super class). Thus multiple-inheritance of interfaces is not multiple-inheritance in the Stroustrup sense [Sto 1987].

Our system is like the *JigSaw* system of Bracha in that it has rigorous semantics, base upon a denotional model of delegation [Bracha]. We decouple proxy

delegation from subtyping (unlike the Lieberman-style of delegation). By extending an existing language with a new API to obtain several benefits:

1. An upwardly compatible extension.
2. Realistic performance.
3. A practically useful tool.
4. Polymorphism is obtained by implementing synthesized interfaces.
5. Inheritance is restricted to subtypes only.
6. Name collisions are resolved by topological sorting or programmer interaction.

It has been asserted in the past that refactoring will also be language dependent because it must understand the language of the programs that it is manipulating. I shall show that this is generally untrue. The semi-automatic static proxy delegation system does not use Java source. This approach can be used in any language with a reflection API [Johnson].

2. Delegation vs. Multiple Inheritance

Multiple inheritance is a hotly debated language feature [Tempero]. Multiple inheritance gives us code reuse and polymorphism. Delegation enables code-reuse without polymorphism. The uncertainties that can arise from the use of multiple-inheritance of implementation have been cited as the rationale for leaving this language feature out of Java [Arnold 1998]. Yet, for some reason, inheritance seems to be more popular than delegation. It has been suggested that one reason for this might be that in multiple inheritance, classes transparently inherit operations from their superclasses. Synthesizing delegations automatically reduces the possibility of introducing errors and should encourage programmers to use delegation more [Johnson].

Both delegation and inheritance are mechanisms for extending a design [Coad]. For example, if multiple inheritance of implementations existed in Java, our *MovableMammal* might be written like this:

```
public class MovableMammal extends Movable, Mammal {
}
```

This would work using the multiple-inheritance model of C++. Such a model has been shown to have several disadvantages. For example:

1. Subclasses must inherit only a single implementation from a super class.
2. The topological sorting of the super-classes have been cited as a fruitful source of bugs [Arnold 1996].
3. Inheritance compromises the benefits of encapsulation [Coad].
4. Inheritance hierarchy changes are unsafe [Snyder].
5. Even in a single-inheritance type language like Java, conflicts between multiple parents are not reported. Ambiguity resolution has long been known as a problem with inheritance [Kniesel].
6. Taxonomically organized data has become automatically associated with *object-oriented programming* [Cardelli].

Some have said that multiple inheritance is hard to implement, expensive to run and complicates a programming language [Cardelli]. These conjectures are shown to be generally untrue by Stroustrup [Stro 1987].

Multiple inheritance provides for subtyping. It is this feature that the *interface* mechanism of Java embraces by providing multiple inheritance of *specification* without *implementation*. Thus an interface *x* is a subtype of interface *y* if *x* is a descendant of *y*. This also works for classes, in Java, but such relationships are subject to only single-inheritance.

Systems, like *Kiev*, extend the Java language so that it has multiple inheritance <http://www.forestro.com/kiev/kiev.html>. Sorry to say, that makes for a rather non-standard and unportable solution (as opposed to the one presented here). The

LAVA language also extends Java to provide for delegation. Kniesel says that current implementations of LAVA have an efficiency that is unacceptable [Kniesel 98] [Kniesel 99]. Fisher and Mitchell provide a new delegation-based language [Fisher]. Sorry to say, the language is untyped and therefore has the unsoundness of any dynamic delegation system. The primary advantage of the Fisher-Mitchell system appears to be in resolving method name conflicts at compile-time (something that most multiple-inheritance systems fail to do).

Reverse engineering programs, such as *Lackwit*, are able to discover inheritance relationships with greater ease than composition associations [O'Callahan]. That is because the inheritance association implies a specialization semantic.

Specialization is as legitimate an object-oriented design technique as composition or aggregation. For example; is *polluted water* a kind of water, or is it water that *has* pollution in it? Or do we say that *polluted water is made of water and pollution*? That is, do we use aggregation, composition or specialization to model polluted water? Since all three representations are legitimate models of polluted water, depending on the application, all should be permitted.

On the other hand, specialization is often an inadequate way of modeling associations [Frank]. For example, roles in a multiple-inheritance structure may change. This is the notion of changing *roles*. For example, an insurance company sees the children of clients as *dependents* in its software system. However, after the children grow up they can change from dependents to *clients*. In a static multiple inheritance relationship, role changing is not easy. This is a failure to model dynamic evolution of the world [Kniesel]. Thus, in the example of the role, we delegate to role instances that represent kinds of roles that a person may have. Frank suggests the association of *acts-as* be used for various kinds of roles. For example, a *person acts-as* a *student* [Frank].

Delegation has been cited as a mechanism to obtain implementation inheritance via composition [Lie 1986], [Jz 1991]. Delegation was introduced in a prototype-based object model by Lieberman in 1986 [Lie 1986]. Lieberman indicated that delegation is considered safer than inheritance because it forces the programmer to select which method to use when identical methods are available in two delegate classes. Thus, any means of synthesizing code that contains methods with identical signatures, will cause syntax errors. These errors require the programmer to *think* about which method to use, rather than use automatic mechanisms based on topological sorting of the super classes. Topological sorting (as in C++ and ZetaLisp) has been shown to be a fruitful source of bugs in multiple-inheritance type languages. This is why it has been omitted from languages like Modula-3, Objective C and Java [Har] [Cox].

We are motivated to automate the gathering of the implementations from a collection of instances and place them into a proxy class. This is called message forwarding and is a kind of implementation sharing mechanism [Kniesel]. Experts have disagreed on this point, saying that delegation is a form of class-inheritance (since the execution context must be passed to the delegate). I take the opposite view, as class-inheritance type of sharing of context involves name sharing, property sharing and method sharing. Sharing via delegation is instance sharing. The semantics of instance sharing enable a control of the coupling between instances. This provides a mechanism for reuse without introducing uncontrolled cohesion (which increases brittleness in the code) [Bardou].

Delegation has the disadvantage that:

1. The computational context must be passed to the delegate.
2. There is no straightforward way for the delegate to refer back to the delegating object [Viega].
3. The proxy cannot override the delegate methods automatically (that requires programmer intervention).
4. The proxy class is coupled to the delegates.

With JDK 1.3, there is a new technique called *dynamic proxies* [Sun 2000]. Dynamic proxies have all the disadvantages of delegation and:

5. They are harder to understand than more static software.
6. Dynamic delegation is slower than static delegation.
7. The design has a counterintuitive class structure [Korman]
8. Type-safe dynamic delegation is impossible [Kniesel 98].

Point 8 requires some discussion. Dynamic proxies do not give you compile-time checking of unresolved messages. In contrast, static delegation does provide compile-time checking of unresolved messages. This is a critical difference. Even multiple-inheritance will compile-time check unresolved messages. Thus, in the spectrum of type-safety, we have, in order of most-safe first:

1. Static delegation
2. Multiple-inheritance
3. Dynamic proxy classes

The multiple-inheritance is less type-safe than the static delegation because method ambiguity is typically resolved, without warning, at compile time. Thus, some unexpected behavior can result. We enable automatic disambiguation in the proxy class by virtue of topological sorting. However, this is only one option. We also enable a programmer selection via a GUI so that disambiguation can occur in a semi-automatic manner. Finally, we can also output ambiguous code so that the programmer can resolve the ambiguities at compile-time. The last technique is probably the least reliable method, since the programmer is in the code synthesis loop.

Kniesel has defined delegation as having automatic method forwarding (i.e., Lieberman delegation). We prefer to use the term *dynamic delegation*. The static method forwarding (which Kniesel says is not “true” delegation) is what I define as static delegation [Kniesel 99] [Kniesel 01]. Static delegation is type-safe, dynamic

delegation is not. The methods invoked remain the same, but the change in behavior comes from a change in *implementation*.

Stroustrup tried an implementation of dynamic delegation in C++. He reported that every user of the delegation mechanism "suffered serious bugs and confusion". He says that the primary reasons are that functions in the proxy do not override functions in the delegate and functions in the delegate can not get back to the proxy (i.e., the *this* is in a different context). Stroustrup mentions a solution, by manually forwarding a request to another object (i.e., static delegation) [Stro 1994].

The static delegation we propose is able to alter its behavior in a type-safe way, at run-time. To show that semi-automatic static proxy delegation will change the behavior, depending on the instance of the delegate, one need only to make use of polymorphic delegates. A proxy that interfaces to a numerical computing toolkit, for example, could take an instance that defines a function. Naturally, the toolkit would be useless if it only worked for a single function. A different function will cause different behavior in the toolkit. Yet the function always conforms to an interface that requires a double precision number on input and output. Thus, our system is a type-safe example of proxy delegation.

Manual delegation has the disadvantage that:

1. Tedious wrappers need to be written for each method.
2. Manually writing forwarding methods is error-prone.
3. Programmers write arbitrary code in a forwarding method. This can give an object an inconsistent interface.
4. Programmers must decide which message subset must be forwarded.

Automatic proxy delegation overcomes these problems.

1. The delegation synthesis does not generate arbitrary code.
2. The interface to the instances remains consistent.

3. The delegation is subject to in-line expansion and is more efficient than multiple inheritance.
4. The mechanism for forwarding is obvious and easy to understand.
5. The proxy is coupled to the delegate in a more controlled manner than automatic dynamic delegation.
6. Classes that use the proxy are presented with a more stable interface than the proxy class. For example, a method may become deprecated, but changes need only be seen in the proxy class, not its clients.

The additional advantage is that we can lower the cost of software maintenance and improve reusability of the code.

Problems that remain unsolved by automatic static proxy delegation include:

1. Programmers can write arbitrary code in a forwarding method.
2. There is no straightforward way for the delegate to refer back to the delegating object [Viega].
3. Programmers could limit the forwarding message subset (i.e., make the proxy into a facade).
4. The computational context must still be passed to the delegate [Kniesel].
5. The proxy class is fragile. If the interface to the delegate changes, the forwarding method in the proxy must change [Kniesel 1998].

An additional step, the compilation of generated code, is required with static proxy delegation (automatic or manual). This compilation step disambiguates. In comparison, dynamic proxy classes generate error at run time. We favor compile-time errors over runtime errors, and so find our technique superior in this regard.

Semi-automatic synthesis of delegation code addresses the time-consuming and error-prone draw-back of manual delegation. It is also easier to understand. The basic issue is that a balance must be struck between code reuse and the fragility that arises from *coupling*, a measure of component interdependency. This balance is

obtained by good object-oriented design (and is hard to automate!). We follow some basic rules, suggested by Stroustrup when implementing C++ classes, for the synthesis of our proxy classes:

1. Ambiguities are illegal.
2. Only public methods are available
3. The methods are all public in the proxy class.
4. Subtyping is done with interfaces, not proxies.
5. Both proxy classes and interfaces are synthesized automatically.
6. Type checking is static.
7. Ambiguity resolution is static (i.e., done at code synthesis time).

When ambiguities arise, the synthesizer resolves them using either topological sorting or by interacting with the programmer. Thus the code is synthesized without ambiguities. If ambiguities existed in the output code, it would cause a syntax error. Our code, once compiled, will guarantee that we will never get messages like “can’t find method” [Wand].

Multiple inheritance of interfaces enables the use of *polymorphism* just like multiple inheritance of classes. In Java, *interfaces* are used as a means to achieve polymorphism. Typically, the interfaces are coded manual, and as they change, so too must the implementing classes. I call this the fragile interface problem. It is generally solved by adding new interfaces, and new classes, without altering old ones. The problem with this approach is that the new interface cannot make old methods more restrictive in their access. Even worse, as we elect to shift our code to the new interface, often we are performing maintenance, in parallel, on identical implementations. We solve the problem of implementation reuse by automating the proxy class generation. We solve the problem of creating the new interfaces (and detecting name collisions) using the reflection API. This enables the automatic synthesis of interfaces based on sample instances and allows us to achieve polymorphism. For example, the following code was generated automatically:

```
interface MammalMovableStub extends
```

```
      MammalStub, MovableStub {
}
 interface MammalStub {
      public boolean isHairy();
 }
 interface MovableStub {
      public void move(int v0,int v1);
 }
```

There is a trade-off between using a *pure* proxy and a hybrid proxy, that obtains some of the methods from the single-inheritance model of Java. There are several difficulties with this, for example; a change in the constructor in the proxy class will require that all the constructors in the super class be repeated.

For example, consider the *RunButton*, a class that knows how to run itself, using the command pattern [Gamma 95]:

```
package gui;

import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public abstract class RunButton extends
        JButton implements ActionListener, Runnable {
        public RunButton(String label) {
               this(label,null);
        }
         public RunButton(String l, Icon i) {
                super(l,i);
                addActionListener(this);
        }
        public RunButton(Icon i) {
               this(null,i);;
        }
        public RunButton() {
                this(null,null);
        }
        public void actionPerformed(ActionEvent e) {
               run();
        }
        public static void main(String args[]) {
          ClosableJFrame cf = new ClosableJFrame("Run Button");
          Container c = cf.getContentPane();
          c.add(new RunButton("OK") {
                     public void run() {
```

```
                        System.out.println("I am
        running!");
                    }
                }
          );
         }
   }
```

Now we see, from the above example, that adding the single method of *addActionListener* to just one element of the constructor causes all the constructors to be repeated. While use of the *RunButton* is now more elegant, there is no clean divorce between subtyping and implementation. The single inheritance model does not scale well. Even worse, name conflicts are resolved in silence by the single inheritance mechanism in Java. Thus, the exact method to be invoked is no longer statically known when the code is compiled [Bracha]. Our goal is to provide an alternative for eliminating the naming conflicts, and divorce the subtyping and implementation inheritance. In section 4 we show how to add methods and change usage by creating a new proxy-decorator class.

3. Related work

Tools for refactoring code automatically are not new [Opdy92b], [Opdy93a], [John93b]. Language independent tools for refactoring code are not new either [Tichelaar]. Even the use of explicit and parametrical bindings to create type-safe inheritance is not new [Hauck].

However, in the literature that we have reviewed, we have yet to find a means for automatically creating the proxy classes shown in this paper. In addition, the tools that we have found for refactoring code are like the *Elbereth* system in that they require source code [Korman]. Other source code based tools for automatic refactoring include the Smalltalk Refactoring Browser [Roberts], the IntelliJ Renamer (http://www.intellij.com), which supports renaming of identifiers and the *Xref-Speller* (http://www.xref-tech.com/speller/) which supports set refactorings. The Daikon invariant detector reads source code and depends on instrumentation of

the source code for full function (http://sdg.lcs.mit.edu/~mernst/daikon/). None of the afore mentioned tools automate proxy class synthesis. This is also true for the class composition proposed by Harrison and Ossher [Harrison]. Our technique does not require any source code, but our technique can still generate it.

Our technique for static delegation requires that every instance be passed to a proxy-class, along with its execution context. Thus a programmer's updates in the protocol for communicating the means to pass parameters will have to be updated in the proxy class. This is the solution I took in *Java Digital Signal Processing* [Lyon 1998]. The trouble is, the updates for the interfaces to the delegates may change. This requires proxy class maintenance.

When the proxy protects the client from changes in the delegate specifications, the proxy is making use of the *adapter* pattern. If additional responsibilities are added to the wrappers around the delegates, we are using the *decorator* pattern [Gamma 1995]. Once manual additions are made to the proxy class, it can no longer be regenerated automatically without a loss of the changes. Thus, adding additional responsibilities to an instance of a proxy should probably be left to a new, decorator class. Similarly, if a totally new interface is needed in the proxy class, a new adapter class should be constructed. If a sub-set of methods is needed to simplify the subsystem use, then a *facade* class should be created, to interface to the proxy class. It is generally poor design to make the proxy both the facade, adapter and decorator (though this is likely to be the case, as the code evolves over time).

When a language, like Java, lacks multiple inheritance, we can only rely upon single inheritance or delegation as a means toward code reuse. The drawback of delegation is the constant updating of the proxy code needed to communicate the computing context to the delegate. Our means of automating the synthesis of delegation code is like the pre-processor approach of the *Jamie* system used by [Viega]. A problem with *Jamie* is that it extends the language by creating a macro-preprocessor. Aside from JSP technology, Java has no macros. This enables

symbolic debuggers to work directly with source code that has been seen by (and perhaps, written by) humans.

Jamie provides a means for performing *dynamic* delegation. This is inherently less efficient that static multiple inheritance and static multiple inheritance is less efficient (and less safe) than static delegation. Our technique of semi-automatic static proxy delegation enables inlining of code so that invocations are expanded, something *Jamie* and the dynamic proxy classes of Jdk 1.3 cannot do.

The use of reflection to automatically generate *static* delegation code, even if the original source code is unavailable, is new. This can assist in the creation of facades and toolkits [Gamma 1995].

4. A Real-example

In this section we describe an example of the *Proxy* class that is generated by the *DelegateSynthesizer* and the *ReflectUtil* class. The effect is to alter the interface to the delegates so that it is simpler to use, without having to change any of the existing code. For example, in order to use the *ReflectUtil* and the *DelegateSynthesizer* in the past, we would write:

```
public static void main(String args[]) {
  DelegateSynthesizer ds = new DelegateSynthesizer();
  ReflectUtil ru = new ReflectUtil(ds);
  ds.add(new java.util.Vector());
  ds.process();
  System.out.println(
    ds.getClassString());
}
```

Now we write:

```
public static void main(String args[]) {
  Proxy p = new Proxy();
  p.add(new Vector());
  p.process();
  System.out.println(
    p.getClassString());
}
```

The *Proxy* class contains all the methods of the *ReflectUtil* class and the *DelegateSynthsizer* class, with a different constructor than either of the two delegates. The constructor was coded by hand, and the class was renamed. Other than that, the code output by:

```
public static void main(String args[]) {
  DelegateSynthesizer ds = new DelegateSynthesizer();
  ReflectUtil ru = new ReflectUtil(ds);
  ds.add(ds);
  ds.add(ru);
  ds.process();
  System.out.println(
    ds.getClassString());
}
```

was all that was required to construct the *Proxy* class. We are now able to get an automatically generated interface, called the *ProxyStub* by executing:

```
public static void main(String args[]) {
  Proxy p = new Proxy();
  p.add(p);
  p.process();
  System.out.println(p.getInterfaces());
}
```

This enables us to obtain the multiple-inheritance of typing that we would otherwise have missed if we used only delegation. The *Proxy* class can now implement the *ProxyStub*. In fact, the methods of any number of instances can be folded into a synthesized interface.

5. The Delegate Synthesizer

This section details the implementation of the automatic proxy code synthesis via reflection. Reflection enables a listing of methods and their signatures. These are used to *forward* invocations to the delegates contained by a *proxy* class. I call this *static proxy delegation*, in order to differentiate it from the *dynamic proxy classes* that have been introduced in JDK 1.3 [Sun 2000].

The *DelegateSynthesizer* class generates Java source code that automatically wrappers all the invocations to a list of delegate instances. As an example, consider the programmer who establishes a set of classes based on mammals and humans:

```
class Mammal {
      public boolean isHairy() {
      return true;
      }
}
class Human extends Mammal {
      public String toString() {
        return "human";
      }
}
```

This seems like standard stuff. A *Human* is a kind of *Mammal*. As a result, the *extends* shows a sub-classification of the *Mammal* class. Also, we recognize that the cardinality of the set of all mammals is smaller than the set of all humans. Thus, *extends* represents a kind of knowledge about taxonomic hierarchies.

Suppose we want graphics in our system. We define a new class called *Movable*:

```
class Movable {
      int x = 0;
      int y = 0;

      public void move(int _x, int _y) {
        x = _x;
        y = _y;
      }
}
```

To add features to the *Movable* class, we create the *Graphics* class:

```
class Graphic extends Movable {
      public void erase() {
        move(-1,-1);
      }
}
```

The sub-classification of *Mammal* has a different *intention* than the extension of the *Movable* class. We only extended *Movable* to inherit an implementation, *not to describe a taxonomy!*

In order to obtain a graphic-human (i.e., a class that represents a human that can be drawn) we require delegation. To invoke the *DelegateSynthesizer* we write:

```
Vector v = new Vector();
v.addElement(new Human());
v.addElement(new Graphic());
DelegateSynthesizer ds = new DelegateSynthesizer(v);
ds.print();
```

The output follows:

```
// automatically generated by the DelegateSynthesizer
public class HumanGraphic  {

// constructor:
public HumanGraphic(
      Human _human,
      Graphic _graphic){
      human = _human;
      graphic = _graphic;
}

 Human human;
      public java.lang.String toString(){
        return human.toString();
      }
      public boolean isHairy(){
        return human.isHairy();
      }
 Graphic graphic;
      public void move(int v0,int v1){
        graphic.move(v0,v1);
      }
      public void erase(){
        graphic.erase();
      }
}
```

Thus, the *HumanGraphic* class has all the *public* methods in both the *Human* class and the *Graphic* class.

Several policy decisions were made during the generation of the proxy class. First, it was decided that only public methods would be exposed using this technique. Second it was decided that the base *java.lang.Object* class should be eliminated from the generated proxy class. Thus, no member variables are made visible in the

generated proxy class. Also, any *native*, *abstract*, or *final* modifiers are removed. Finally, any *static* declarations are redeclared to be dynamic, since an instance of the delegate is required for the proxy to work.

5.1. Implementation of the *DelegationSynthesizer*

The implementation of the *DelegationSynthesizer* was filled with special cases of various string manipulation and reflection routines. The goal is to provide support for a GUI that enables the programmer to disambiguate conflicting method signatures. Figure 5.1-1 shows an image of a GUI that enables the programmer to disambiguate conflicting method names using a manual selection technique.

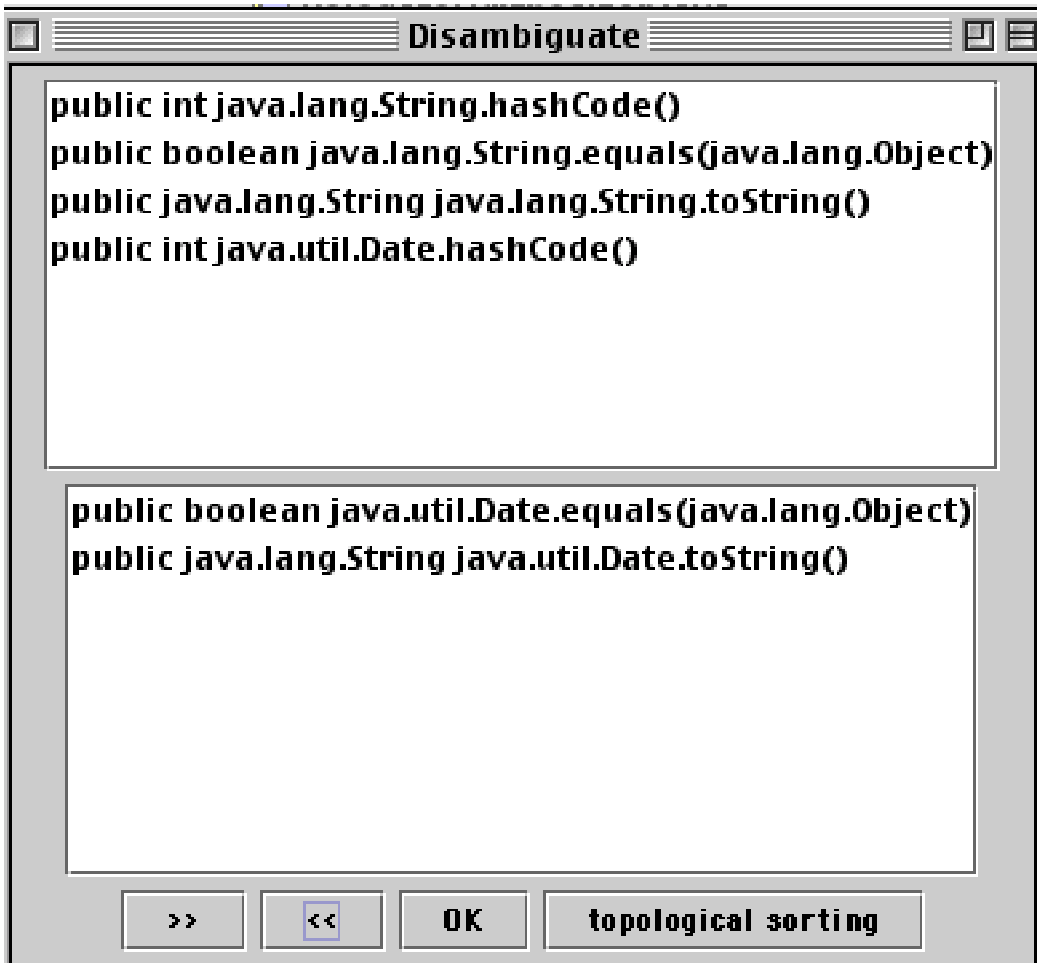

Figure 5.1-1. The Disambiguation Dialog

The *disambiguation dialog* will generate a proxy class using either topological sorting or manual selection. In fact, if the preference is toward manual selection, there is no need for a GUI at all (we can let the process work automatically). If the programmer would like to select the methods to be used, some sort of GUI eases the process.

Several string manipulation procedures are used to simplify the presentation of instances and methods. For example, to strip the package name from a string we use:

```
public static String stripPackageName(String s) {
  int index = s.lastIndexOf('.');
  if (index == -1) return s;
  index++;
  return s.substring(index);
}
```

Proper naming is also required. For example the default string representation of the name of an array of classes is:

```
[Ljava.lang.Class
```

To get the type name to look like an array, we use:

```
public static String getTypeName(Class type) {

  if (! type.isArray())
    return type.getName();

  Class cl = type;
  int dimensions = 0;
  while (cl.isArray()) {
      dimensions++;
      cl = cl.getComponentType();
  }
  StringBuffer sb = new StringBuffer();
  sb.append(cl.getName());

  for (int i = 0; i < dimensions; i++)
      sb.append("[]");

  return sb.toString();
}
```

This yields types like:

```
java.lang.Class[]
```

To get the parameters for a method we use:

```
public String getParameters(Method m) {
  StringBuffer sb = new StringBuffer("");
  Class[] params = m.getParameterTypes(); // avoid clone
     for (int j = 0; j < params.length; j++) {
     sb.append(
```

```
                getTypeName(params[j])+ " v"+j);
            if (j < (params.length - 1))
                sb.append(",");
          }
          return sb.toString();
        }
```

This allows for multiple parameters with synthesized variable names, like:

```
        public java.lang.reflect.Method getMethod(
            java.lang.String v0,java.lang.Class[] v1){
          return class.getMethod(v0,v1);
        }
```

To get the parameters for a lengthy delegation constructor we use:

```
        public String getConstructorParameters() {
          StringBuffer sb = new StringBuffer("\n\t");
          for (int i=0; i < instanceList.size(); i++) {
            ReflectUtil ru = new ReflectUtil(
                instanceList.elementAt(i));
            String instanceName =

            stripPackageName(ru.getClassName()).toLowerCase();
            sb.append(ru.getClassName()
                + " _"
                + instanceName
            );
            if (i < instanceList.size() - 1)
                sb.append(",\n\t");
          }
          return sb.toString();
        }
```

This allows us to get the parameters of a constructor formed from the *Class* class and the *ReflectUtil* class:

```
   // constructor:
   public ClassReflectUtil(
         java.lang.Class _class,
         ReflectUtil _reflectutil){
         class = _class;
         reflectutil = _reflectutil;
   }
```

The constructors' body is obtained using:

```
        private String getConstructorBody() {
          StringBuffer sb = new StringBuffer("\n\t");
```

```
 for (int i=0; i < instanceList.size(); i++) {
   ReflectUtil ru = new ReflectUtil(
        instanceList.elementAt(i));
   String instanceName =

   stripPackageName(ru.getClassName()).toLowerCase();
   sb.append(
        instanceName
        + " = _"
        + instanceName
        + ";"
   );
   if (i < instanceList.size() - 1)
        sb.append("\n\t");
 }
 return sb.toString();
}
```

5.2. The DelegateSynthesizer

The code for the *DelegateSynthesizer* follows:

```
1.    import java.lang.reflect.*;
2.    import java.util.*;
3.
4.    public class DelegateSynthesizer {
5.      private String className = "";
6.      private String methodList = "";
7.      private Vector instanceList ;
8.
```

The *Vector* holds a list of instances that are processed during the construction.

```
9.      public DelegateSynthesizer(Vector _instanceList) {
10.         instanceList = _instanceList;
11.         for (int i=0; i < instanceList.size(); i++)
12.             processInstance(instanceList.elementAt(i));
13.     }
14.
```

The *getConstructorParameters* returns a string that will pass in the instances to be used for the delegation. The instance names are formulated by taking the class names, converting them to lower case and prepending them with an "_" character.

```
15.     public String getConstructorParameters() {
```

```
16.         StringBuffer sb = new StringBuffer("\n\t");
17.         for (int i=0; i < instanceList.size(); i++) {
18.             ReflectUtil ru = new ReflectUtil(
19.                 instanceList.elementAt(i));
20.             String instanceName =
21.
        stripPackageName(ru.getClassName()).toLowerCase();
22.             sb.append(ru.getClassName()
23.                 + " _"
24.                 + instanceName
25.             );
26.             if (i < instanceList.size() - 1)
27.                 sb.append(",\n\t");
28.         }
29.         return sb.toString();
30.     }
```

The *getConstructorBody* performs a series of operations so that the delegates are internally held by the synthesized class. The member variable names are formulated by using the class names, without the package prefix, after conversion to lower case.

```
31.     private String getConstructorBody() {
32.         StringBuffer sb = new StringBuffer("\n\t");
33.         for (int i=0; i < instanceList.size(); i++) {
34.             ReflectUtil ru = new ReflectUtil(
35.                 instanceList.elementAt(i));
36.             String instanceName =
37.
        stripPackageName(ru.getClassName()).toLowerCase();
38.             sb.append(
39.                 instanceName
40.                 + " = _"
41.                 + instanceName
42.                 + ";"
43.             );
44.             if (i < instanceList.size() - 1)
45.                 sb.append("\n\t");
46.         }
47.         return sb.toString();
48.     }
49.
```

The *processInstance* is invoked by the constructor. It synthesizes the proxy class using a name that concatenates the class names, without the package names.

```
50.     private void processInstance(Object o) {
```

```
51.         ReflectUtil ru = new ReflectUtil(o);
52.         String cn = stripPackageName(ru.getClassName());
53.         String instanceName = cn.toLowerCase();
54.         className = className +
55.             stripPackageName(cn);
56.         Method m[] = ru.getAllMethods();
57.         methodList = methodList
58.             + " "+ ru.getClassName() +" "+ instanceName +
       ";\n"
59.             + getMethodList(m,instanceName);
60.     }
```

The *getMethodList* uses an array of methods to create the Java code needed to delegate to an instance.

```
61.     public String getMethodList(Method m[],String
       instanceName) {
62.         String s = "";
63.         for (int i=0; i < m.length; i++)
64.             s = s + getMethodDeclaration(m[i],
       instanceName) + "\n";
65.         return s;
66.     }
```

The *getMethodDeclaration* works on public methods. The declaration makes use of parameters and an instance that is to serve as the delegate for the implementation of the method.

```
67.     public String getMethodDeclaration(Method m, String
       instanceName) {
68.         if (isPublic(m))
69.             return "\t"
70.                 + "public" // strip out other modifiers.
71.                 + " "
72.                 + getReturnType(m)
73.                 + " "
74.                 + m.getName()
75.                 + "("
76.                 + getParameters(m)
77.                 + "){\n\t"
78.                 + getInvocation(m,instanceName)
79.                 + "\t}";
80.         return "";
81.     }
```

The *getReturnType* uses reflection to obtain the type of the return. It then uses *getTypeName* to map that type into a string. The string papers over the string normally returned to make it Java compatible.

```
82.     public static String getReturnType(Method m) {
83.         return  getTypeName(m.getReturnType());
84.     }
85.
```

The *isReturningVoid* method returns *true* if the method, indeed, returns *void*.

```
86.     public static boolean isReturningVoid(Method m) {
87.         return getReturnType(m).startsWith("void");
88.     }
```

The *getModifiers* method returns a string of all the modifiers (i.e.,public static abstract, etc.).

```
89.     public static String getModifiers(Method m) {
90.         return Modifier.toString(m.getModifiers());
91.     }
```

When formulating the delegation, we must not return in the body of the delegation method, if the method returns *void*. Thus, we get a string that represents an *optional* return:

```
92.     private String getOptionalReturn(Method m) {
93.         if (isReturningVoid(m)) return "";
94.         return "return ";
95.     }
```

The *getInvocation* class synthesizes a series of variables, *v0, v1, v2...* which are used when formulating the delegation.

```
96.     private String getInvocation(Method m, String
       instanceName) {
97.         StringBuffer sb = new StringBuffer(
98.             "\t"
99.             + getOptionalReturn(m)
100.                    + instanceName
101.                    + "."
102.                    + m.getName()
103.                    + "("
104.                );
105.                Class[] params = m.getParameterTypes();
106.
107.                for (int j=0; j < params.length; j++) {
```

```
108.                    sb.append("v"+j);
109.                    if (j < (params.length - 1))
110.                        sb.append(",");
111.                }
112.                sb.append(");\n");
113.                return sb.toString();
114.            }
```

The *getParameters* method is used by the *getMethodDeclaration* to obtain the arguments to the proxy classes' presentation to the outside word. If you wanted to add exception handling by finding out what exceptions are thrown in a given method, this is a place to add it.

```
115.            public String getParameters(Method m) {
116.                StringBuffer sb = new StringBuffer("");
117.                Class[] params = m.getParameterTypes();
// avoid clone
118.                for (int j = 0; j < params.length; j++) {
119.                    sb.append(
120.                        getTypeName(params[j])+ " v"+j);
121.                    if (j < (params.length - 1))
122.                        sb.append(",");
123.                }
124.                return sb.toString();
125.            }
126.
```

The *getTypeName* is a helper method that maps the type into a name that can be compiled by Java. The work occurs when the type is not an array.

```
127.            public static String
                    getTypeName(Class type) {
128.
129.                if (! type.isArray())
130.                    return type.getName();
131.
132.                Class cl = type;
133.                int dimensions = 0;
134.                while (cl.isArray()) {
135.                    dimensions++;
136.                    cl = cl.getComponentType();
137.                }
138.                StringBuffer sb = new StringBuffer();
139.                sb.append(cl.getName());
140.
141.                for (int i = 0; i < dimensions; i++)
142.                    sb.append("[]");
```

```
143.
144.                 return sb.toString();
145.             }
146.
147.
```

The *isPublic* method is used to determine if a method will be used for delegation. The policy is that only public methods will be available for delegation. If you wanted to change this policy, you could create an *isPublicOrDefault* method. This would return *true* if default visibility were to be passed to delegates. This might be used to transform methods with default visibility into *public* methods for inter-package communication via a *facade*. See Section 1.4 for more information about facades.

```
148.             public static boolean isPublic(Method m) {
149.                 return
150.                 Modifier.toString(
                    m.getModifiers()).startsWith("public");
151.             }
```

We lop off the package name to create instance names for the delegates. This is done via a simple string manipulation. A more robust approach might be used that senses if the resulting string, when converted to lower-case, results in a reserved word. In such cases, a simple alteration to the string is in order and this might be a good place to do it.

```
152.             public static String stripPackageName(String
       s) {
153.                 int index = s.lastIndexOf('.');
154.                 if (index == -1) return s;
155.                 index++;
156.                 return s.substring(index);
157.             }
```

The first pass, performed during construction, instantiates a series of member variables in the *DelegateSynthesizer*. These variables are used to permit the creation of the constructor body. If this code were to be optimized, it would be to transform it into a single pass process. As it is, speed is not a factor.

```
158.             private String getConstructor() {
```

```
159.                // public className(class1
      _class1Instance, class2 _class2Instance...) {
160.                //    class1Instance = _class1Instance;
161.                //  class2Instance = _class2Instance;
162.                //}
163.                return "\n// constructor: \npublic "
164.                    + className
165.                    + "("
166.                    + getConstructorParameters()
167.                    +"){"
168.                    + getConstructorBody()
169.                    + "\n}\n\n";
170.            }
```

The *getClassString* method is the top-level means for generating the proxy class. If the output is to be retargetted to a live, on-line, compiler, or a swing interface, or a file, this would be the method to call. It can easily be wrappered so that the code generated can be redirected.

```
171.            public String getClassString() {
172.                 return
173.  "// automatically generated by the DelegateSynthesizer"
174.                    +"\npublic class "
175.                    + className
176.                    + " {\n"
177.                    + getConstructor()
178.                    + methodList
179.                    + "}\n";
180.            }
181.            public void print() {
182.                print(getClassString());
183.            }
184.            private void print(Object o) {
185.                System.out.println(o);
186.            }
187.
188.
```

The following code example generates a proxy for the *DelegateSynthesizerReflectUtil* class. This class combines the public methods of both classes. Thus the *DelegateSynthesizer* can generate delegates for itself.

```
189.            public static void main(String args[]) {
190.                Vector v = new Vector();
191.                v.addElement(
                     new DelegateSynthesizer(v).getClass());
```

```
192.                 v.addElement(new ReflectUtil(v));
193.                 v.addElement(new DelegateSynthesizer(v));
194.                 DelegateSynthesizer ds =
                      new DelegateSynthesizer(v);
195.                 ds.print();
196.             }
197.     }
```

6. Conclusions

We have reviewed different techniques for adding features to classes. We discussed using language extension to add delegation, language extension to add multiple inheritance, API extension to add delegation and API extension to add manual delegation. Approaches that use language extension fail for pragmatic reasons (lack of compatible tools, slow adoption, slow code, etc.). Approaches that use API extension are easier to deploy, in general, since they work with existing frameworks.

There are two basic kinds of delegation, dynamic and static. The dynamic delegation works at run-time and makes type safety impossible. The static delegation works at compile time and is generally type-safe. There are two kinds of static delegation, manual and automatic. The manual delegation requires programmers to generate method forwarding code. A process that is both error-prone and tedious. The automatic static delegation has been shown to be an easy to deploy technique that gives programmers the freedom to generate large proxy classes that are both type-safe and easy to understand.

The automatic synthesis of proxy class technique makes modification of the method forwarding mechanics trivial. It also isolates client code from changes in the interfaces in the delegates (called the *adapter pattern*). The adapter controls the brittleness of a subsystem from propagating to client classes. As changes (i.e., deprecations) are introduced into an API, the rest of the system can remain unchanged. The synthesis technique shown here allows for incrementally checking

the synthesized code in an interactive system. It also allows for the resolution of ambiguity using topological sorting in an automatic fashion.

There are several problems with the automatic delegation code generated from reflection. Exception handling will have to be explicitly supported, if desired, inside of the delegate method. It was not clear that the throws clause should be added to the methods that throw exceptions, or if they should be handled locally. As a result, this was left to the programmer.

Any methods that are commonly held in two or more delegate instances will have a name-space conflict. It is not clear how to automatically resolve these conflicts. If we use topological sorting, like C++ does, we may have the same fruitful source of bugs that C++ has. As a result, we have adopted a policy that programmers' must decide how to resolve conflicting method names. This is not easy. For example, if the code generated removed duplicate method declaration, what would be a reasonable policy to use? Suppose that a *Human* knows how to print itself, using a *print* statement. Suppose that the *Graphic* class has a *print* statement. It might be reasonable to print both delegates. On the other hand, suppose both have a *toString* method. In that case, it might be reasonable to concatenate the *toString* results from both instances. Thus, the programmer must intervene to correct the duplication of method signatures.

If the name of a class has an upper-case version of a reserved word (like the class *Class*), then the delegate synthesizer will generate code that will fail to compile. That is because it is not *smart* enough to know the reserved word when it sees one. Aside from making the code smarter, a simple replace-string (executed by the programmer) will fix the problem.

The *facade* design pattern creates a single class that communicates with a collection of related classes. For example, *ReflectUtil, Class, DelegationSynthesizer* are all related to introspection. The *DelegationSynthesizer* can be used to create a proxy

class that delegates to itself, and the *ReflectUtil and Class* classes. Client instances need only be aware of the new *ReflectUtilClassDelegationSynthesizer*. This protects the clients from changes in the API (i.e., deprecation). It can also shield the clients from the complexity of using the individual classes. This is particularly true if the programmer takes care to simplify the constructor of the facade. This also prevents knowledge of the order of construction from being required in the client classes. If the specification changes on the delegate, and the proxy alters the forwarding methods to protect the client classes from the change, then the proxy ceases to be a simple proxy. The proxy becomes an example of the *adapter* pattern, stabilizing the interface seen by the clients [Gamma 1995].

The approach to delegation using static binding enables inlining of code. The inlining optimizes the code by eliminating the forwarding methods, when possible, by the compiler. Thus static delegation does not suffer from performance degradation, like dynamic delegation does.

In brief:

1. Dynamic delegation is more automatic than static delegation.
2. Dynamic delegation is not type-safe, but static delegation is.
3. Automatic static delegation is almost as automatic as dynamic delegation, and just as type safe as static delegation.

The choid between static and dyanmic typing is like the choice between safety and flexibility. [Agesen].

Multiple-inheritance is not an option in Java. The following are some heuristics for the use of the approach outline in this paper:

If polymorphism is needed, then use the automatically generated interface stubs, that our API provides.

If proxies are needed, then use our API for generating proxies.

If source code is unavailable, there may be little other choice.

If source code is available, refactoring by hand may lead to better code, but may have an effect on a large number of client classes and require testing.

If many programmers require a stable interface, then use the automatically generated interface stubs, and create a facade for a contract that enables control and use of the subsystem.

Use the proxy to reuse the implementations. In the case where the contracts shift in the delegates, allow the facade to become an adapter-facade-proxy, in order to protect your clients.

Deepening subclasses in order to add features is a fast way to create poor code that is very fragile. It is a poor way to introduce sub-typing. Only use subclasses if the class theoretic approach is appropriate to the domain, and then only if the taxonomic hierarchy is unlikely to change.

7. Future Work

In the future we would like to expand the *DelegateSynthesizer* so that methods that require exception handling are thrown in the proxy class.

It would improve the generated code if the *DelegateSynthesizer* if it included the Javadoc from the delegate's method in its output code. This would enable a preservation of the input documentation. Unfortunately, access to the source code would be required for this to work.

A proxy class helps to isolate a system from deprecations in the delegate methods. Sun's repeated introduction of deprecation into its API's has become epidemic and is a topic of future research.

8. Literature Cited

[Agesen] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. “Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance”. In *ECOOP '93 Conference Proceedings*, p. 247-267. Kaiserslautern, Germany, July 1993

[Aksit 1991] Mehmet Aksit, Jan Willem Dijkstra. “Atomic Delegation: Object-oriented transactions”, *IEEE Software*, Los Alamitos, CA, IEEE Computer Society. March 1991.pps. 84-92.

[Arnold 1996] Ken Arnold and James Gosling. *The Java Programming Language*, Addison-Wesley, Reading, MA. 1996.

[Arnold 1998] Ken Arnold and James Gosling. *The Java Programming Language*, Second Edition, Addison-Wesley, Reading, MA. 1998.

[Bardou] D. Bardou and C. Dony. “Split Objects: A Disciplined Use of Delegation Within Objects”. In Proceedings of OOPSLA'96, Sans Jose, California. Special Issue of ACM SIGPLAN Notices (31)10, pages 122-137, 1996. http://citeseer.nj.nec.com/bardou96split.html

[Bracha] G. Bracha. “The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance”. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992

[Brant] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. “Wrappers to the Rescue”. In Proceedings of ECOOP'98, July 1998. http://citeseer.nj.nec.com/189005.html

[Booch 1991] Grady Booch. *Object-Oriented Design*, Benjamin Cummings, Redwood Cits, CA. 1991.

[Cardelli] L. Cardelli, “Semantics of Multiple Inheritance”, *Information and Computation*, 76 (1988) 138-164.
<http://citeseer.nj.nec.com/cardelli88semantics.html>

[Coad] Peter Coad and Mark Mayfield. “Java-Inspired Design: Use Composition Rather than Ineritance”, *American Programmer*, Jan. 1997, pps. 23-31.

[Compagnoni] Compagnoni, A. B., & Pierce, B. C. 1993 (Aug.). “Multiple Inheritance via Intersection Types”. *Tech. rept. ECS-LFCS-93-275. LFCS*, University of Edinburgh. Also available as Catholic University Nijmegen computer science technical report 93-18.
<http://citeseer.nj.nec.com/compagnoni93multiple.html>

[Cox] B. Cox. “Message/Object Programming: An evolutionary change in programming technology”, *IEEE Software* (1)1. Jan 1982.

[Fisher] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In Proc. of FCT, volume 965 of Lecture Notes in Computer Science, pages 42--61. Springer-Verlag, 1995. <http://citeseer.nj.nec.com/104746.html>

[Frank] Ulrich Frank. “Delegation: An Important Concept for the Appropriate Design of Object Models”, *Journal of Object Oriented Programming*, June 2000. pps. 13-17,44

[Fraser] Timothy Fraser, Lee Badger, and Mark Feldman. “Hardening COTS Software with Generic Software Wrappers”. In IEEE Symposium on Security and Privacy, May 1999. http://citeseer.nj.nec.com/fraser99hardening.html

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns*, Addison-Wesley, Reading, MA. 1995.

[Har] S. Harbison. *Modula-3*. Prentice Hall, 1992.

[Hauck] F. J. Hauck: "Inheritance modeled with explicit bindings: an approach to typed inheritance"; *Proc. of the Conf. on Object-Oriented Progr. Sys., Lang., and Appl*. -- OOPSLA, (Washington, D.C., Sep. 26-Oct. 1, 1993); SIGPLAN Notices 28(10) , <http://citeseer.nj.nec.com/hauck93inheritance.html>

[Harrison] William Harrison, Harold Ossher and Peri Tarr, Using Delegation for Software and Subject Composition, Research Report RC 20946, IBM Thomas J. Watson Research Center, August 1999.
<http://www.research.ibm.com/sop/soppubs.htm>

[John93b] Ralph E. Johnson and William F. Opdyke, “Refactoring and Aggregation”, Object Technologies for Advanced Software - First JSSST International Symposium, Lecture Notes in Computer Science, Vol. 742, Springer-Verlag, 1993.

[Johnson] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In S. Nishio and A. Yonezawa, editors, International Symposium on Object Technologies for Advanced Software, pages 264--278, Kanazawa, Japan, November 1993. JSSST, Springer Verlag, Lecture Notes in Computer Science.
<http://citeseer.nj.nec.com/johnson93refactoring.html>

[Jz 1991] Johnson and Zweig. Delegation in C++, *Journal of Object-Oriented Programming*, 4(11):22-35, November 1991.

[Kataoka] Yoshio Kataoka and Michael D. Ernst and William G. Griswold and David Notkin, “Automated Support for Program Refactoring using Invariants” <citeseer.nj.nec.com/kataoka01automated.html>.

[Kniesel] Günter Kniesel: “Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems”. Technical report IAI-TR-94-3, Oct. 1994, University of Bonn, Germany.
<http://citeseer.nj.nec.com/kniesel95implementation.html>

[Kniesel 98] Günter Kniesel: “Delegation for Java: API or Language Extension?”. Technical report IAI-TR-98-5, May, 1998, University of Bonn, Germany. <citeseer.nj.nec.com/kniesel97delegation.html>

[Kniesel 99] Günter Kniesel, “Type-Safe Delegation for Run-Time Component Adaptation”, In R. Guerraoui (Ed.): Proceedings of ECOOP99. Springer LNCS 1628. <http://citeseer.nj.nec.com/kniesel99typesafe.html>

[Kniesel 01] Günter Kniesel, private e-mail communications, <kniesel@cs.uni-bonn.de>.

[Korman] W. Korman and W. G. Griswold. “Elbereth: Tool support for refactoring Java programs”. Technical report, University of California, San Diego Department of Computer Science and Engineering, May 1998. http://citeseer.nj.nec.com/korman98elbereth.html

[Lie 1986] Henry Leiberman. Using prototypical objects to implement share behavious in object-oriented systms. In *Object-oriented Programming Systems, languages and Applications Conference Proceedings*, Pages 214-223.

[Lyon 1998] Douglas Lyon and Hayagriva Rao. *Java Digital Signal Processing*, M&T Books, NY, NY. 1998.

[Lyon 1999] Douglas Lyon. *Image Processing in Java*, Prentice Hall, M&T Books, NY, NY. 1998.

[O’ Callahan] O'Callahan,R., and Jackson, D., “Lackwit: A program understandingtool based on type inference”. In Proceedings of the 1997 International Conference on Software Engineering (ICSE'96) (Boston, MA, May 1997), pp. 338--348. <http://citeseer.nj.nec.com/329620.html>

[Opdy92b]William F. Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. dissertation, University of Illinois, 1992.
[ftp://st.cs.uiuc.edu/pub/papers/refactoring/]

[Opdy93a] William F. Opdyke and Ralph E. Johnson, “Creating Abstract Superclasses by Refactoring”, Proceedings CSC'93, ACM Press, 1993.

[Postema] Margot Postema and Heinz W. Schmidt, *Reverse Engineering and Abstraction of Legacy Systems*, <http://citeseer.nj.nec.com/151140.html>

[Roberts] Don Roberts, John Brant, and Ralph Johnson. “A refactoring tool for Smalltalk” *Theory and Practice of Object Systems*, 3(4):253-63, 1997.

[Tichelaar] “Sander Tichelaar and Stéphane Ducasse and Serge Demeyer and Oscar Nierstrasz, “A Meta-model for Language-Independent Refactoring”, IEEE Proceedings ISPSE, 2000, <citeseer.nj.nec.com/379788.html>

[Snyder] Alan Snyder. “Encapsulation and Inheritance in Object-Oriented Programming Languages”, Affiliation Software Technology Laboratory, Hewlett-Packard Laboratories, PO Box 10490, Palo Alto, CA, 94303-0971 <http://citeseer.nj.nec.com/328789.html>

[Stro 1987] Bjarne Stroustrup. “Multiple inheritance for C++”. In Proceedings of the Spring '87 European Unix Systems User's Group Conference, Helsinki, Finland, May 1987. <http://citeseer.nj.nec.com/stroustrup99multiple.html>

[Stro 1994] Bjarne Stroustrup. *The Design and Evolution of C++*, Addison-Wesley, Reading, MA. 1994.

[Sun 2000] Tech Tips, “Using dynamic proxies to layer new functionalirty over existing code” May 30, 2000,
<http://developer.java.sun.com/developer/TechTips/2000/tt0530.html>

[Sun 2001] "The JavaBeans Runtime Containment and Services Protocol specification" May 24, 2001, <http://java.sun.com/products/javabeans/glasgow/#containment>.

[Tempero] Ewan Tempero and Robert Biddle, "Simulating mulitiple inheritance in Java", The Journal of Systems and Software 55(2000) pps. 87-1000, Springer-Verlag.

[Vega] John Viega and Bill Tutt and Reimer Behrends, "Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages", CS-98-03, Microsoft Coporation, Feb., 1998, <http://citeseer.nj.nec.com/3325.html>

[Wand] Mitchell Wand. "Type inference for record concatenation and multiple inheritance". In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92--97, Pacific Grove, CA, June 1989. http://citeseer.nj.nec.com/wand89type.html